Spring 3-1-1990

An Abstract Data Type for Name Analysis ; CU-CS-460-90

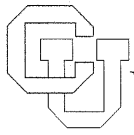
Kastens
University of Colorado Boulder

William M. Waite
University of Colorado Boulder

An Abstract Data Type for Name Analysis

U. Kastens
W. M. Waite

CU-CS-460-90



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

An Abstract Data Type for Name Analysis

U. Kastens
W. M. Waite

CU-CS-460-90 March, 1990

ABSTRACT: This paper defines an abstract data type on which a solution to the name analysis subproblem of a compiler can be based. We give a state model for the ADT, and showed how that model could be implemented efficiently. The implementation is independent of any particular name analysis, so it is possible to create a library module that can be used in any compiler. Such a library module has been incorporated into the Eli compiler construction system.

1. Introduction

The problem of compiling a program can be decomposed into a number of subproblems, many of which have standard solutions. If a subproblem has a standard solution, there is no need for a compiler designer to re-invent that solution. The standard solution can be packaged in a module and re-used. Development of a library of such modules should be accorded high priority by researchers studying the compiler construction process.

In this paper we consider the name analysis subproblem. This is the problem of determining which source language entity is denoted by each identifier occurrence in a program. For example, in the Pascal program of Figure 1 the programmer has used the identifier *A* to denote a variable on line 10. On line 12, *A* denotes a field of the record pointed to by *P*. The identifier *B*, on the other hand, has been used to denote a field of the record *R* (declared on line 6) on both lines. A Pascal compiler's name analysis task must determine which entity is denoted by each occurrence of *A*, which by each occurrence of *B*, and so forth.

Source language entities are described by sets of *properties*. The Pascal entity declared on line 6 of Figure 1 is a variable capable of holding values of a particular type. Its lifetime is the entire execution history of the program, and it will occupy a certain amount of storage at a particular address during execution. Thus this entity might be described by the following property values:

```
 1   program Context (input ,output );
 2   const C =1.2;
 3   type T =record A ,B ,C : integer end;
 4   var
 5     A : real;
 6     R : T;
 7     P : ↑T;
 8   begin
 9   new (P ); readln (R.B ,R.C ,P ↑.C );
10   A :=R.B +C ;
11   with P ↑ do (* Re-defines A B C *)
12     A :=R.B +C ;
13   R :=P ↑; writeln (A ,R.A );
14   end.
```

Figure 1
A Pascal Program

Class:	Variable	
Type:	*real*	(determines the storage requirement also)
Level:	1	(indicates the variable is global)
Offset:	-12	(address relative to the global storage area base)

All entities would have a *Class* property in their description, but the remaining properties might be different for entities of different classes.

Each entity can be characterized by a *key*, which allows the compiler to access the entity's properties. Because the key characterizes the source program entity, the name analyzer need only determine a key for each identifier occurrence. Property access may be implemented by an appropriate data base technique or a definition table with one entry for each key, but these decisions are totally independent of name analysis.

Name analysis is based on the *scope rules* of the source language being compiled. Scope rules are formulated in terms of *definitions*, which cause identifiers to denote source language entities, and *program regions*: For each definition, the scope rules specify the program region in which that definition is valid. In Figure 1, the variable declaration on line 6 is a definition that causes the identifier R to denote a particular Pascal entity. The scope rules of Pascal state that this definition is valid for the region consisting of lines 2 through 14.

A definition is represented in the compiler by a binding between an identifier and a key. The particular set of (identifier,key) bindings valid at a program point is called the *environment* of that program point.[1] At the occurrence of A in line 10 of Figure 1, the scope rules of Pascal specify that bindings exist for C, T, A, R, P, and all of the predefined identifiers of Pascal. At the occurrence of B on the same line, however, only bindings for A, B and C are valid. Moreover, the binding for A valid at the occurrence of A is different from the binding for A valid at the occurrence of B.

The concept of an environment can be captured in an abstract data type (ADT). This ADT provides operations for creating environments, populating them with (identifier,key) bindings, and carrying out mappings from identifiers to keys based upon them. If a module that efficiently implements the environment ADT is available, a compiler's name analysis task can be carried out by appropriate invocations of that module's operations. The module itself is independent of the source language; different scope rules affect only the way in which the module is invoked.

We describe the structure and operations of the environment ADT algebraically in Section 2. Section 3 presents a state model of the ADT, and Section 4 shows how that model can be efficiently implemented by a standard module. In Section 5 we show how the standard environment module can be used to carry out name analysis for several important kinds of scope rules, demonstrating that it supports a wide range of compilers.

2. The Environment Abstract Data Type

The computations used during name analysis can be specified algebraically in terms of three abstract data types:

Identifier — The compiler's internal representation of an identifier
DefTableKey — The compiler's internal representation of a key
Environment — The compiler's internal representation of a scope

Each textually-distinct identifier is represented by a unique value of type *Identifier*. (The meaning of "textually-distinct" depends, of course, on the source language. A compiler for Pascal would represent both *Xy* and *xy* by the same *Identifier* value, but a C compiler would use different *Identifier* values for these identifiers.) Values of type *DefTableKey* provide access to the properties of an object. There is a distinguished *DefTableKey* value, *NoKey*, that provides access to no properties. We shall not explore the details of the *Identifier* and *DefTableKey* ADTs in this paper.

There are as many interesting *Environment* values as there are distinct scopes in a program. ("Scope" was introduced in the ALGOL 60 Report[2] as the name for the region of a program in which the declaration of an identifier denoting a particular entity is valid.) In most programs, several definitions are valid over identical regions of the program and thus the number of distinct scopes is smaller than the number of distinct (identifier,key) bindings. For example, in Figure 1 the scopes of the field identifiers *A*, *B* and *C* are identical. The constant, type and variable identifiers in Figure 1 also have identical scopes, and these scopes are distinct from those of the field identifiers. A third set of identical scopes, those for the pseudo-variables implicitly defined by the **with** statement, exists in Figure 1. Finally, the Pascal Standard[3] defines a scope external to the program for the required identifiers (e.g. *integer*, *real*). Thus four *Environment* values would be used by a Pascal compiler when compiling Figure 1 because there are four distinct scopes in that program.

Although Figure 1 has only four distinct scopes, there are five distinct environments for program points. Let us consider two of these:

1) The environment for the occurrences of *B* in lines 10 and 12, and the second occurrence of *A* in line 13, contains only bindings for *A*, *B* and *C*. Each identifier is bound to a key characterizing a field of the record type declared on line 3.

2) The environment for the program points between the keywords **record** and **end** in line 3 contains the bindings of environment (1), plus bindings for *T*, *R*, *P* and all of the predefined identifiers of Pascal.

The difference between these two environments is that (2) has inherited a number of bindings valid in the region outside the record declaration. That inheritance is the result of the following scope rule, first stated for ALGOL 60 and since used for almost all hierarchically-structured programming languages with static binding:

> A definition is valid in its own scope and in all nested scopes not containing a definition for the same identifier.

As a consequence, a definition in a scope hides definitions for the same identifier in enclosing scopes, which is why environment (2) did not inherit the binding between the

identifier *A* and the key for the real variable declared on line 5.

Environments (1) and (2) are represented by the same *Environment* value. The distinction is embodied in the environment abstract data type by preserving the hierarchical relationships among the *Environment* values and providing two access operations. One of these access operations yields only the bindings in the scope corresponding to a specified *Environment* value. That access function would be used to implement environment (1). The other access function, used to implement environment (2), yields the bindings in the scope corresponding to a specified *Environment* value and any bindings from enclosing scopes (provided those bindings are not hidden).

Figure 2[4] describes the behavior of the environment abstract data type in terms of *Identifier* and *DefTableKey*. The scope hierarchy is conveyed to the ADT through use of *NewScope*, whose operand is the *Environment* value for the scope enclosing the one being created. *KeyInScope* makes available only the bindings whose scope is specified by its first operand, while *KeyInEnv* makes available the bindings in that scope and all enclosing scopes. Note that the hiding is accomplished by searching the nest of scopes hierarchically, from smallest to largest, and returning the first binding found.

Both *KeyInScope* and *KeyInEnv* are total functions returning a single value of type *DefTableKey*. Some languages, for example Algol 68[5] and Ada,[6] allow an environment

---

Signatures

NewEnv:		→	*Environment*
NewScope:	*Environment*	→	*Environment*
Add:	*Environment* × *Identifier* × *DefTableKey*	→	*Environment*
KeyInScope:	*Environment* × *Identifier*	→	*DefTableKey*
KeyInEnv:	*Environment* × *Identifier*	→	*DefTableKey*

Axioms

(A1)	*KeyInScope* (*NewEnv* (),*i*)	=	*NoKey*
(A2)	*KeyInScope* (*NewScope* (*e*),*i*)	=	*NoKey*
(A3)	*KeyInScope* (*Add* (*e*,i_1,*k*),i_2)	=	**if** i_1=i_2 **then** *k* **else** *KeyInScope* (*e*,i_2)
(A4)	*KeyInEnv* (*NewEnv* (),*i*)	=	*NoKey*
(A5)	*KeyInEnv* (*NewScope* (*e*),*i*)	=	*KeyInEnv* (*e*,*i*)
(A6)	*KeyInEnv* (*Add* (*e*,i_1,*k*),i_2)	=	**if** i_1=i_2 **then** *k* **else** *KeyInEnv* (*e*,i_2)

Figure 2
Algebraic Specification of an Environment ADT

---

to contain more than one binding for an identifier. An identifier is *overloaded* in an environment if that environment contains more than one binding for the identifier. The key denoted by an overloaded identifier is determined by applying *overload resolution* rules to the set of bindings in the environment. We shall return to the question of overloading in Section 5, where we show how to handle overloaded identifiers with a standard environment module.

There are two situations in which a compiler would create a new *Environment* value by adding a binding:

1) An identifier definition has been imported from some other context. In this case, both the identifier and the key to which it is to be bound are known.

2) A new identifier definition has been encountered. In this case, the identifier is known but the compiler must obtain a new key from the *DefTableKey* ADT.

Regardless of which of these two situations arise, the compiler must deal with the possibility that a declaration for the identifier already exists in the current scope. Because these situations arise frequently, and because each is always handled in exactly the same way, it is convenient to augment the environment ADT with explicit operations to handle them (Figure 3).

If there is no binding for i in the current scope e, then each of the operations described by Figure 3 introduces one. *AddIdn*, which would be used in situation (1), binds i to the key k. *DefineIdn* binds i to a new key that it obtains by invoking the

---

Signatures

AddIdn:	*Environment*×*Identifier*×*DefTableKey*	→	*Boolean*×*Environment*
DefineIdn:	*Environment*×*Identifier*	→	*DefTableKey*×*Environment*

Axioms

(A7) *AddIdn*(*e*,*i*,*k*) = **if** *KeyInScope*(*e*,*i*)=*NoKey*
then (*true*,*Add*(*e*,*i*,*k*))
else (*false*,*e*)

(A8) *DefineIdn*(*e*,*i*) = **if** *KeyInScope*(*e*,*i*)=*NoKey*
then (*k*,*Add*(*e*,*i*,*k*)) **where** *k*=*NewKey*
else (*KeyInScope*(*e*,*i*),*e*)

Figure 3
Operations to Introduce Bindings

NewKey operation of the *DefTableKey* abstract data type. The new key is returned so that the compiler can set its properties. Since the key supplied to *AddIdn* has presumably already had properties set, *AddIdn* simply returns *true* to indicate that the definition was allowed.

If the current scope e already contains a binding for i, *AddIdn* reports that fact by returning *false*. *DefineIdn*, on the other hand, conceptually maps all definitions of the same identifier in one scope to the same key. Hence the situation in which an identifier is multiply defined has to be indicated by a property associated with that key. In any event, only the first definition of an identifier in a scope is retained; later definitions of that identifier are ignored.

Figure 4 shows how the environment ADT might be used to perform name analysis. *Identifier* values are denoted by the identifiers themselves in Figure 4, *DefTableKey* values by k_n and *Environment* values by e_n. The invocations of ADT operations are written next to the constructs with which they are associated. Thus the compiler must execute a *DefineIdn* operation for the construct in the first line of each program.

The operations invoked in Figures 4a and 4b are identical, but their arguments differ because of the differing scope rules of C and Pascal. In C, a definition is valid from the point of definition to the end of the compound statement containing that definition, while in Pascal a definition is valid throughout the block containing that definition. This difference in scope rules is manifested in the *Environment* argument of the *KeyInEnv* invocation for i in the definition of m in each figure. In Figure 4a, e_5 is used and in Figure 4b e_6 is used. The difference between these two values is that e_5 contains only the bindings that textually precede the use of i in the declaration of m, whereas e_6 contains all of the bindings in the entire block.

Note that the order of the invocations of the ADT operations in Figure 4 is determined by the dependences among them, not by the order in which they are written down. All of the *DefineIdn* invocations in Figure 4b must be executed before any of the *KeyInEnv* invocations, because the latter require e_6 as an argument.

Figure 4b contains an error, because the use of i in the declaration of m precedes its declaration in the following line. The Pascal Standard prohibits use of an identifier before its declaration in all cases except the declaration of a pointer type. Many Pascal compilers do not detect this error: The dependences among the operations in Figure 4b imply that the compiler must visit all identifier definitions in a block before visiting any identifier uses. A compiler that is designed to make exactly one pass, in textual order, over the source program will not be able to satisfy the dependences of Figure 4b without retaining a representation of the entire tree in memory. (There is a way to use the name analysis strategy of Figure 4a for Pascal, keeping additional properties to detect the use-before-definition error.[7])

A straightforward implementation of the environment ADT uses a single record for each invocation of *NewScope* and *Add*. Each *Environment* value is a pointer to one of these records. The record corresponding to an operator invocation simply contains all of the arguments of that invocation. Since *Environment* values are pointers, this means that the records form singly linked lists that can be searched recursively by procedures that

```
int i = 10;                          DefineIdn(e1,i)→k1,e2
P()                                  DefineIdn(e1,P)→k2,e3
{                                    NewScope(e3)→e4
   int m = i;                        DefineIdn(e4,m)→k3,e5
                                     KeyInEnv(e5,i)→k1
   int i = 20;                       DefineIdn(e5,i)→k4,e6
   printf("%d %d\n", m, i);          KeyInEnv(e6,printf)→k5
                                     KeyInEnv(e6,m)→k3
                                     KeyInEnv(e6,i)→k4
}
```

a) According to the scope rules of C

```
const i = 10;                        DefineIdn(e1,i)→k1,e2
procedure P;                         DefineIdn(e2,P)→k2,e3
                                     NewScope(e3)→e4
   const m = i;                      DefineIdn(e4,m)→k3,e5
                                     KeyInEnv(e6,i)→k4
   i = 20;                           DefineIdn(e5,i)→k4,e6
   begin
   writeln(m, i);                    KeyInEnv(e6,writeln)→k5
                                     KeyInEnv(e6,m)→k3
                                     KeyInEnv(e6,i)→k4
   end;
```

a) According to the scope rules of Pascal

Figure 4
Name Analysis Using the ADT of Figure 3

exactly implement the axioms A1-A6 of Figure 2.

Each operation except *NewScope* has a worst case time complexity of $O(N)$ in this implementation, where N is the number of identifier declarations in the program. Since the total number of identifier occurrences is proportional to the number of identifier declarations, and some operation with time complexity $O(N)$ must be invoked for every identifier occurrence (*AddIdn* or *DefineIdn* for definitions, *KeyInEnv* or *KeyInScope* for

uses), the asymptotic time complexity of name analysis as a whole will be $O(N^2)$. The asymptotic storage requirement is $O(N)$ because there is one fixed-length record for each identifier definition (records corresponding to *NewScope* operations could be eliminated by adding two flags to the records corresponding to *Add* operations).

Another possible implementation uses an array of records, indexed by *Identifier* values, for each *Environment* value. Each record contains a *DefTableKey* value (possibly *NoKey*) specifying the binding of its index, and a Boolean value that is true if and only if the binding occurred in the scope represented by the array. *KeyInEnv* and *KeyInScope* are constant time operations in this implementation. Unfortunately the asymptotic space complexity is $O(N^2)$, and the total time complexity for name analysis remains $O(N^2)$: Both *NewScope* and *Add* must create new arrays, copying the contents of their *Environment* arguments and setting the flags properly. Thus each of these operations has time complexity $O(N)$, and the number that must be executed is proportional to N.

3. A State Model for the Environment ADT

The algebraic specification of the environment ADT given in the previous section has a strict value semantics. Each application of a constructor function (*NewEnv*, *NewScope*, *Add*) yields a new *Environment* value that exists from that time forward. Figure 4 shows, however, that each of these values has a specific "useful lifetime": the portion of the execution history of the name analysis between the time the value is created and the time it is last used. This suggests that the definitions of the previous section overspecify the environment ADT, and that by making a specification that is more precise in terms of the lifetimes of *Environment* values we might be able to reduce the cost of name analysis.

The useful lifetime of an *Environment* value depends upon the name analysis strategy used by the compiler, which in turn depends on the scope rules of the source language. Figure 4 illustrated the effect of the essential difference in scope rules: For some languages the scope of a definition begins at the defining point and continues to the end of a region (*C-like scope rules*); for others the scope of a definition is the entire region in which it is declared (*Pascal-like scope rules*). A compiler for a language with C-like scope rules might carry out its name analysis task during a single text-order traversal of the source program, as illustrated by Figure 4a. Compilers for languages with Pascal-like scope rules, on the other hand, might traverse every region of the program twice. During the first traversal they would invoke *AddIdn* or *DefineIdn* at every definition and ignore all nested regions. The second traversal would invoke *KeyInEnv* or *KeyInScope* at every use of an identifier, and would perform both traversals of each nested region. (This strategy is compatible with the value dependence of Figure 4b. We have already pointed out the existence of strategies that retain additional information in order to avoid two traversals when compiling Pascal.)

Now consider the effect of these strategies on the useful lifetimes of the *Environment* values generated by the environment ADT described in the previous section. *NewEnv* will be invoked at the beginning of the compilation, and *NewScope* will be invoked at the beginning of the traversal of each nested region. *KeyInEnv* or *KeyInScope* will be invoked at each use of an identifier. Clients of the abstract data type never

invoke *Add* directly; they add bindings by invoking either *AddIdn* or *DefineIdn* when a definition is encountered. It is easy to show that if an *Environment* value is used by either *AddIdn* or *DefineIdn*, that value will never be used again by either of the strategies discussed in the previous paragraph. We will therefore assume that *AddIdn* and *DefineIdn* do not return new *Environment* values, but instead alter the state of their *Environment* argument by adding a binding to the set it represents. (Of course these operations will have no effect on the state if a binding for their *Identifier* argument is already present in their *Environment* argument.) An *Environment* value's state is nothing but the set of bindings it represents. Thus the environment of a program point is actually the state of some *Environment* value.

Figure 5 shows the consequences of this state model for the examples of Figure 4. Notice that it is no longer possible to determine the order of the invocations from dependences among them. The compiler designer must explicitly take the state of the *Environment* value into account when deciding upon the correct invocation sequence. In Figure 5a the invocation of *KeyInEnv* for i in the line defining m must precede the invocation of *DefineIdn* for i in the following line because the latter operation will change the state of e_2. Similarly, the invocation of *KeyInEnv* for i in the line defining m in Figure 5b must follow the invocation of *DefineIdn* for i in the following line because only after the latter operation has the proper state of e_2 been established.

Let us now consider an implementation of the state model using an array of fixed-size records to implement each *Environment* value. The state of the value defines the content of the array. Each *NewEnv* and *NewScope* operation has time complexity $O(N)$, because it must create a new array and either initialize the contents (*NewEnv*) or copy the content of another array (*NewScope*). All other operations, however, require only constant time. *DefineIdn* and *AddIdn* check a single array element and possibly alter its content; *KeyInScope* and *KeyInEnv* simply access the element. The overall time and space complexity of name analysis are therefore reduced to $O(N \times S)$, where N is the number of identifier occurrences and S is the number of distinct regions of the program in which identifiers are declared.

4. A Standard Environment Module

This section presents an implementation of a standard environment module that supports name analysis in a wide range of compilers. It realizes the state model of the environment ADT described in the previous section. The basic approach is to use a single array of lists of fixed-size records to simulate the implementation discussed at the end of the previous section. For practical situations both the time and space complexity of name analysis using this module are $O(N)$. We first give the details of the module and then analyze its performance.

Figure 6a gives a Pascal definition of the data types manipulated by the module, and Figure 6b illustrates the state of the data structure in a Pascal compiler during the processing of line 12 of Figure 1. (Space limitations only permit us to show three of the *StkElt* records; there are actually many more.) The dotted rectangle at the right of Figure 6b is the *AccessMechanism* record, which contains the array and a pointer to the record representing the *Environment* value currently encoded in that array. If the *Environment*

`int i = 10;`	$DefineIdn\,(e_1,i)\rightarrow k_1$
`P()`	$DefineIdn\,(e_1,P)\rightarrow k_2$
`{`	$NewScope\,(e_1)\rightarrow e_2$
`  int m = i;`	$DefineIdn\,(e_2,m)\rightarrow k_3$
	$KeyInEnv\,(e_2,i)\rightarrow k_1$
`  int i = 20;`	$DefineIdn\,(e_2,i)\rightarrow k_4$
`  printf("%d %d\n", m, i);`	$KeyInEnv\,(e_2,printf)\rightarrow k_5$
	$KeyInEnv\,(e_2,m)\rightarrow k_3$
	$KeyInEnv\,(e_2,i)\rightarrow k_4$
`}`	

a) According to the scope rules of C

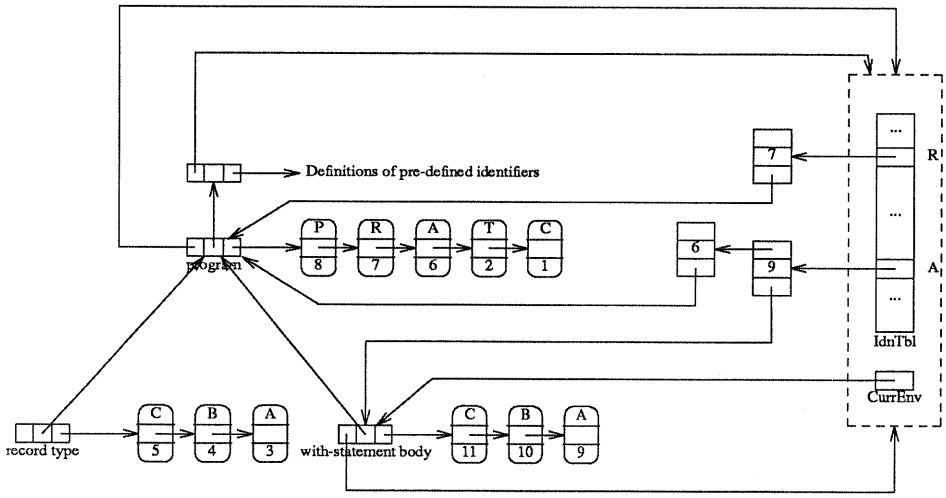
`const i = 10;`	$DefineIdn\,(e_1,i)\rightarrow k_1$
`procedure P;`	$DefineIdn\,(e_1,P)\rightarrow k_2$
	$NewScope\,(e_1)\rightarrow e_2$
`  const m = i;`	$DefineIdn\,(e_2,m)\rightarrow k_3$
	$KeyInEnv\,(e_2,i)\rightarrow k_4$
`  i = 20;`	$DefineIdn\,(e_2,i)\rightarrow k_4$
`  begin`	
`  writeln(m, i);`	$KeyInEnv\,(e_2,writeln)\rightarrow k_5$
	$KeyInEnv\,(e_2,m)\rightarrow k_3$
	$KeyInEnv\,(e_2,i)\rightarrow k_4$
`  end;`	

a) According to the scope rules of Pascal

Figure 5
Name Analysis Using A State Model

```
type
  Environment=↑EnvImpl;          (* Set of Identifier/Definition pairs *)
  Scope=↑RelElt;                 (* Single region *)
  Access=↑AccessMechanism;       (* Current access state *)
  EnvImpl=record                 (* Addressing environment *)
    nested: Access;                (* Constant-time access to identifier definitions *)
    parent: Environment;           (* Enclosing environment *)
    relate: Scope;                 (* Current region *)
    end;
  RelElt=record
    idn: Identifier;               (* Identifier *)
    nxt: Scope;                    (* Next identifier/definition pair for the current region *)
    key: DefTableKey;              (* Definition *)
    end;
  StkPtr=↑StkElt;                (* List implementing a definition stack *)
  StkElt=record
    out: StkPtr;                   (* Superseded definitions *)
    key: DefTableKey;              (* Definition *)
    e: Environment;                (* Environment containing this definition *)
    end;
  AccessMechanism=record         (* Current state of the access mechanism *)
    IdnTbl = array                 (* Stacks of definitions *)
      [0..MaxIdn] of StkPtr;
    CurrEnv: Environment;          (* Environment represented by the array state *)
    end;
```

a) Data objects for the constant-time access function



b) The data structure during the analysis of line 12, Figure 1

Figure 6
The Data Structure of the Standard Environment Module

value passed to one of the module's operations is equal to the value specified by this pointer, then the operation is carried out immediately. Otherwise, the contents of the array may be adjusted to reflect the *Environment* value passed.

The *nested* field of the *EnvImpl* record reflects the actual scope nesting, relative to the scope represented by the *EnvImpl* record. If the scope pointed to by the *CurrEnv* field of the *AccessMechanism* is either the scope represented by the *EnvImpl* record or a scope nested within that scope, then the *nested* field contains a pointer to the *AccessMechanism*. Otherwise the *nested* field contains **nil**. It is easy to see that definitions for an environment are on the "identifier stacks" addressed by *IdnTbl* elements if and only if the *nested* field of its *EnvImpl* record points to the *AccessMechanism*. In Figure 6b, *CurrEnv* is pointing to the *EnvImpl* record for the **with** statement body, which is the environment appropriate for line 12 of Figure 1. The identifiers defined in the **with** statement, the identifiers defined in the program block, and the predefined identifiers are on the identifier stacks. Therefore the *nested* fields of the *EnvImpl* records corresponding to those three regions point to the access mechanism. The *nested* field of the *EnvImpl* record corresponding to the body of the record declared on line 3 of Figure 1 is **nil** because the field identifiers are *not* on the identifier stacks.

Figure 7 gives Pascal code that implements the environment access functions *DefineIdn* and *KeyInEnv*. Both begin by making certain that the array reflects the situation in the environment specified by the *Environment* argument. *SetEnv*, *EnterEnv* and *LeaveEnv* are all private procedures of the environment module. They are used to set up the array so that it reflects a given environment, and are only invoked if some sort of change is necessary.

EnterEnv assumes that the array reflects the parent of the environment to be entered. It scans that environment, pushing a new definition onto the stack for each identifier in the environment. *LeaveEnv* reverses this operation, removing the top definition from the stack for each identifier in the current environment and thus restoring the array to reflect the parent of the current environment. *SetEnv* decides on a sequence of leave and enter operations that will make the array reflect the desired environment. If definitions for the desired environment are on the identifier stacks, but the desired environment is not the current one, *SetEnv* simply leaves environments until the desired environment is reached. Otherwise, *SetEnv* sets the environment to the parent of the desired one and then enters the desired environment.

What is the time complexity of the implementation shown in Figure 7? Clearly *DefineIdn* and *KeyInEnv* are $O(1)$ if the array reflects the *Environment* argument. In order to account for the time required to maintain the array's state, we need to examine the global behavior of the name analyzer. The fundamental question that must be answered is "How often is a particular region considered during name analysis?"

Figure 7 will support a name analyzer that shifts its attention arbitrarily among regions. If each node in the tree is visited a fixed number of times, independent of the size of the program, the tree traversal can only enter a particular region a fixed number of times. If the name analyzer only shifts its attention from one region to another when the tree traversal actually moves from one region to another, then *EnterEnv* and *LeaveEnv* will only be executed a fixed number of times for each region. Both *EnterEnv* and

```
procedure EnterEnv(e : Environment);
  (* Make the state of the array reflect e
    On entry-
      The state of the array reflects the parent of e
  *)
  var r : Scope; s : StkPtr;
  begin r :=e↑.relate;
  with e↑.parent↑.nested↑ do
    begin
    while r <>nil do
      begin new(s); s↑.e :=e; s↑.key :=r↑.key; s↑.out :=IdnTbl[r↑.idn]; IdnTbl[r↑.idn]:=s; r :=r↑.nxt; end;
    CurrEnv :=e;
    end;
  end;
procedure LeaveEnv(e : Environment);
  (* Make the state of the array reflect the parent of e
    On entry-
      The state of the array reflects e
  *)
  var r : Scope; s : StkPtr;
  begin r :=e↑.relate;
  with e↑.nested↑ do
    begin
    while r <>nil do begin s :=IdnTbl[r↑.idn]; IdnTbl[r↑.idn]:=s↑.out; dispose(s); r :=r↑.nxt end;
    CurrEnv :=e↑.parent;
    end;
  end;
procedure SetEnv(e : Environment);
  (* Make the state of the array reflect e *)
  begin
  if e=nil then Report(DEADLY,1004(*Invalid environment*),NoPosition,1);
  if e↑.nested=nil then begin SetEnv(e↑.parent); EnterEnv(e) end
  else with e↑.nested↑ do repeat LeaveEnv(CurrEnv) until CurrEnv=e;
  end;
function KeyInEnv(e : Environment; idn : Identifier): DefTableKey;
  begin
  with e↑.nested↑ do
    begin
    if e <>CurrEnv then SetEnv(e);
    if IdnTbl[idn]=nil then KeyInEnv :=NoKey else KeyInEnv :=IdnTbl[idn]↑.key;
    end;
  end;
function DefineIdn(e : Environment; idn : Identifier): DefTableKey;
  var found: boolean; p : DefTableKey; r : Scope; s : StkPtr;
  begin
  with e↑.nested↑ do
    begin
    if e <>CurrEnv then SetEnv(e);
    if IdnTbl[idn]=nil then found :=false else found :=IdnTbl[idn]↑.e=e;
    if found then DefineIdn :=IdnTbl[idn]↑.key
    else
      begin p :=NewKey;
      new(r); r↑.idn :=idn; r↑.key :=p; r↑.nxt :=e↑.rng; e↑.rng :=r;
      new(s); s↑.e :=e; s↑.key :=p; s↑.out :=IdnTbl[idn]; IdnTbl[idn]:=s;
      DefineIdn :=p;
      end;
    end;
  end;
```

Figure 7
Environment Access Operations

LeaveEnv execute a constant number of basic operations for each element of the region's *relate* list, which has one entry for each defining occurrence in the region.

Consider a particular region, R. Suppose that R is entered and left T times. Each time the region is entered, E basic operations are executed for each element of *R.relate* (Figure 6a); each time it is left, L operations are executed for each element of *R.relate*. Therefore the total number of basic operations executed during the entire compilation for each element of *R.relate* would be $T\times(E+L)$. We can therefore "charge" each defining occurrence in R the time required to execute $T\times(E+L)$ basic operations. This shows that we can ignore the costs of *EnterEnv* and *LeaveEnv*, because their only effect is to increase the cost of *DefineIdn* by a constant amount.

All of the operations of the environment module except *KeyInScope* and *NewEnv* are $O(1)$ in the worst case when implemented as shown in Figure 7. *KeyInScope* would also be $O(1)$ if the compiler entered the appropriate region before *KeyInScope* was applied. Then all that would be needed would be to verify that the association at the top of the stack was defined in that region, just as *DefineIdn* does. Taking this approach, however, would mean that applied occurrences of field identifiers in Pascal constructs like *R.B* would require the name analyzer to shift its consideration from the current region to the region of a record definition and then immediately shift it back again. Although Figure 7 is quite capable of doing this, it violates our assumption that the number of times the name analyzer shifts its attention to a given region is independent of the program size. Thus it makes the cost of *EnterEnv* and *LeaveEnv* impossible to ignore.

In order to salvage our earlier analysis, we must charge the cost of an *EnterEnv* operation followed by a *LeaveEnv* operation to the invocation of *KeyInScope* that processes *R.B*. But since that cost is proportional to the number of fields in R, and the cost of *KeyInScope* implemented as a linear search is also proportional to the number of fields in R, nothing has been gained. Since the search operation is simpler than *EnterEnv* followed by *LeaveEnv*, *KeyInScope* should not use the array access mechanism.

For most programs, field references like *R.B* are a small fraction of the total number of identifier occurrences. Moreover, the number of fields in a record does not usually grow with overall program size. We can conclude that the implementation of Figure 7 gives essentially $O(1)$ complexity for every operation of the environment module except *NewEnv*. Since *NewEnv* is $O(N)$ and is executed once, and the other operations are $O(1)$ and executed $O(N)$ times, the actual growth of the name analysis time with program size is $O(N)$. This contrasts with the $O(N^2)$ growth for the implementation of Section 2.

5. Additional Environment Access Operations

The data structure of Figure 6 provides easy access to information beyond that needed to simply map identifiers into keys. For example, to process the **with** statement that accesses a record, a Pascal compiler needs access to a list of all of the fields defined for records of that type. It must create a "pseudo-variable" corresponding to each field. The address of the pseudo-variable is the sum of the address of the record accessed by

the **with** statement and the relative address of the field within that record, and the type of the pseudo-variable is the type of the field. Since every record type is a region containing definitions of all of the fields, such a list must be a component of the standard environment's data structure.

When resolving overloading in Algol68 or Ada, the compiler needs to consider hidden bindings for the overloaded identifiers. Again, the data structure for the standard environment has these lists. By defining additional operations, we can make these lists available at very little cost. It is important to note, however, that the additional operations do *not* form a part of the environment abstract data type and may not be easy to provide with other implementations.

In each case, strict value semantics are appropriate for the operations. One operation returns the desired list, given the *Environment* (or *Environment*×*Identifier*) value on which that list is based. Other operations accept a list of the proper type and yield the information contained in its first element. Finally, an operation accepts a list of the proper type and returns that list with the first element deleted. Figure 8 defines the necessary operations with their signatures. (*Scope* and *StkPtr* are defined in Figure 6a.)

Implementation of the operations of Figure 8 is trivial, provided that we can guarantee that the lists on which they operate will not be altered during their operation. This is a reasonable restriction to impose upon the user, given the tasks for which the operations are intended. The restriction could be checked by the module, but such checking is probably not worthwhile.

6. A More Complex Example

The name analysis examples of Figure 5 illustrate the basic use of the standard environment module. In this section we show an additional example that illustrates how so-called "explicit scope control" is handled.[8] It turns out that some care must be taken

---

DefinitionsOf :	*Environment*	→	*Scope*
IdnOf :	*Scope*	→	*Identifier*
KeyOf :	*Scope*	→	*DefTableKey*
NextDefinition :	*Scope*	→	*Scope*
HiddenBy :	*Environment*×*Identifier*	→	*StkPtr*
HiddenKey :	*StkPtr*	→	*DefTableKey*
NextHidden :	*StkPtr*	→	*StkPtr*

Figure 8
Useful Auxiliary Operations

---

to preserve the $O(N)$ time bound when processing languages with this form of visibility rules.

Figure 9 shows a program fragment written in Modula2.[9] Each of the comments could be replaced by a sequence of statements, and the variables indicated in the comments would be visible in those statements. "Standard identifiers", like *CARDINAL*, are visible everywhere.

The construct "*MODULE X* ··· *END X*;" is called a *module declaration.* A module declaration explicitly controls the visibility of identifiers. Identifiers declared outside a module declaration are not visible inside it unless they are standard identifiers or they have been named in an import list. Identifiers declared inside a module declaration can be made visible outside it if they are included in an export list.

Modula2 also provides procedure declarations, in which the normal Pascal scope rules apply. Thus the name analysis task of a Modula2 compiler builds and uses environments in much the same way as the name analysis task of a Pascal compiler. A module declaration appearing in a region of a program acts as like a sequence of identifier declarations and uses. The name of the module and the identifiers in the export list are all declared by the module declaration, and the identifiers in the import list are used. This effect of the module declaration involves no new concepts, and can be handled by the techniques discussed earlier in this paper.

A module declaration also has the effect of creating a new scope that is a child of the root of the environment module's tree of scopes (see Figure 6b). Identifiers are defined within a scope created by a module declaration in three ways:

```
VAR a,b: CARDINAL;
MODULE M;
  IMPORT a; EXPORT w,x;
  VAR u,v,w: CARDINAL;
  MODULE N;
    IMPORT u; EXPORT x,y;
    VAR x,y,z: CARDINAL;
    (* u,x,y,z are visible here *)
    END N;
  (* a,u,v,w,x,y are visible here *)
  END M;
(* a,b,w,x are visible here *)
```

Figure 9
A Modula2 Program

---

1) They appear in an import list of the module declaration. Identifier *u* is introduced into module *N* in this way (Figure 9).

2) They appear in the export list of a module declared immediately within the module declaration. Identifier *y* is introduced into module *M* in this way (Figure 9). Note that identifier *y* is *not* introduced into the main program because the module *N* is not declared immediately within the main program.

3) They are defined by a declaration immediately within the module declaration. Identifier *v* is introduced into module *M* in this way (Figure 9).

How can we implement these rules using our standard environment module?

For simplicity, we shall assume that the compiler builds a tree that represents the source program. The shape of the tree is determined by the abstract syntax[1] of Modula2. This means, in particular, that each module declaration is represented by a subtree. In the tree representing Figure 9, module *M* is represented by a subtree of the program tree and module *N* is represented by a subtree of the module *M* tree. The compiler traverses the source program tree, visiting nodes and performing computations. Each visit to a specific kind of node is handled by a single *visit procedure*. This visit procedure may perform computations and visit children of its node by invoking the appropriate visit procedures. Ultimately, the visit terminates when the visit procedure returns.

To implement the Modula2 scope rules, we associate two visit procedures with module declarations and two with procedure declarations. Figure 10 gives the interface specifications for the procedures that carry out the first visit, and sketches their algorithms. The effect of these procedures is to traverse the tree depth-first, processing the identifier declarations that occur at the top level in every module. According to the interface specification, this makes the export list of each module accessible via that module's node as a list of (identifier, key) pairs.

It is important to note the difference between *visiting* a construct and *entering the environment* associated with a construct. The routines of Figure 10 visit every procedure and module in the program. Only *ModuleVisit* 1 executes any operations of the environment abstract data type. *AddIdn* verifies that the current environment is the one specified by its *Environment* argument, as shown in Figure 7 for *DefineIdn*. If the current environment is not the desired one, then *AddIdn* invokes *SetEnv* to enter it. Thus the visit procedures of Figure 10 create an environment for every module and enter that environment to define the identifiers declared within it. Note that environment values are *not* created for procedures during these visits.

The procedures that carry out the second visit of each module and procedure are shown in Figure 11. A module visit begins by defining all of the imported identifiers, which are stored (with the appropriate keys) in a list attached to the module's node. This action causes the module's environment to be entered if it is not already current. Then the import lists of all immediately-nested modules are constructed. Since each identifier on an import list of an immediately-nested module is defined in the environment of the current module, no change of environment is required. Next, all of the identifier uses in the module's environment are processed. No change of environment is implied by this processing either.

```
procedure ModuleVisit 1(node : TreePtr );
  (* First visit to a source program tree node representing a local module
    On exit-
      node↑.ExportList =A list of (identifier,key) pairs defining the exported bindings
  *)
  begin
  for n :=(* each immediately-nested module declaration *) do ModuleVisit 1(n );
  for n :=(* each immediately-nested module procedure *) do ProcVisit 1(n );
  node↑.Env :=NewScope (StandardEnvironment );
  for n :=(* each immediately-nested module declaration *) do
    for (i ,k ) in n↑.ExportList do
      if not AddIdn (node↑.Env ,i ,k ) then (* Report an error *);
  (* Process all identifier declarations *)
  node↑.ExportList :=nil;
  for i :=(* each identifier on an export list *) do
    node↑.ExportList :=(i ,KeyInEnv (node↑.Env ,i )) , node↑.ExportList ;
  end;

procedure ProcVisit 1(node : TreePtr );
  (* First visit to a source program tree node representing a procedure
    On exit-
      ModuleVisit 1 has been applied to all local modules
  *)
  begin
  for n :=(* each immediately-nested module declaration *) do ModuleVisit 1(n );
  for n :=(* each immediately-nested module procedure *) do ProcVisit 1(n );
  end;
```

Figure 10
Beginning the Name Analysis for Modula2

```
procedure ModuleVisit2(node : TreePtr);
  (* Second visit to a source program tree node representing a local module
    On entry-
      node↑.ImportList=A list of (identifier,key) pairs defining the imported bindings
  *)
  begin
  for (i,k) in node↑.ImportList do
    if not AddIdn (node↑.Env,i,k) then (* Report an error *);
  for n:=(* each immediately-nested module declaration *) do
    begin n↑.ImportList:=nil;
    for i:=(* each identifier on an export list *) do
      n↑.ImportList:=(i,KeyInEnv (node↑.Env,i)) , n↑.ImportList;
    end;
  (* Process all identifier uses *)
  for n:=(* each immediately-nested module procedure *) do ProcVisit2(n);
  for n:=(* each immediately-nested module declaration *) do ModuleVisit2(n);
  end;

procedure ProcVisit2(node : TreePtr);
  (* First visit to a source program tree node representing a procedure
    On exit-
      ModuleVisit2 has been applied to all local modules
  *)
  begin
  node↑.Env:=NewScope ((*enclosing environment*));
  for n:=(* each immediately-nested module declaration *) do
    for (i,k) in n↑.ExportList do
        if not AddIdn (node↑.Env,i,k) then (* Report an error *);
  (* Process all identifier declarations *)
  (* Process all identifier uses *)
  for n:=(* each immediately-nested module declaration *) do
    begin n↑.ImportList:=nil;
    for i:=(* each identifier on an export list *) do
      n↑.ImportList:=(i,KeyInEnv (node↑.Env,i)) , n↑.ImportList;
    end;
  for n:=(* each immediately-nested module procedure *) do ProcVisit2(n);
  for n:=(* each immediately-nested module declaration *) do ModuleVisit2(n);
  end;
```

Figure 11
Continuing the Modula2 Name Analysis

When an immediately-nested procedure is visited, *ProcVisit* 2 uses *NewScope* to create the environment for that procedure. It then defines the identifiers exported by modules immediately contained within that procedure in the procedure's environment. This causes the procedure's environment to be entered. After processing all of the identifier declarations of the procedure, *ProcVisit* 2 processes the identifier uses and constructs import lists for the immediately-nested modules. None of these operations alter the current environment.

After all of the identifiers that are not included in nested procedures or modules have been processed, there is no need to remain within or return to the environment of the current procedure. Thus if that environment is left, it will never be re-entered. *ProcVisit* 2 now applies itself to each immediately-nested procedure. It creates environments for those procedures, populates them with definitions, and looks up identifier uses within them. Each environment is created and then entered exactly once. Only after all identifier occurrences in a given scope have been dealt with does *ProcVisit* 2 move on to another scope. Finally, *ProcVisit* 2 invokes *ModuleVisit* 2 for each immediately-nested module.

It is easy to see that Figures 10 and 11 enter each module's environment twice (once in *ModuleVisit* 1 and once in *ModuleVisit* 2) and each procedure's environment once (in *ProcVisit* 2). Thus the complexity analysis of Section 4 holds, and Modula2 name analysis is $O(N)$.

The constraint of a fixed number of environment entries and exits can be met with visit sequences other than those implied by Figures 10 and 11. For example, in *ModuleVisit* 1 the **for** statements on the first two lines could be combined into a single traversal that visited all immediately-nested modules and procedures in textual order. When making such simplifications, however, it is important to verify that the constraint is, in fact, met: If we were to combine the **for** statements on the last two lines of *ProcVisit* 2 into a single traversal that visited all immediately-nested modules and procedures in textual order then the constraint would be violated!

References

1. D. A. Schmidt, *Denotational Semantics*, Allyn and Bacon, Newton, MA, 1986.
2. P. Naur, ed., 'Revised Report on the Algorithmic Language ALGOL 60', *Communications of the ACM*, **6**, 1-17 (January 1963).
3. 'Pascal Computer Programming Language', ANSI/IEEE 770 X3.97-1983, American National Standards Institute, New York, NY, January 1983.
4. J. V. Guttag and J. J. Horning, 'The Algebraic Specification of Abstract Data Types', *Acta Informatica*, **10**, 27-52 (1978).
5. A. Wijngaarden, B. J. Mailloux, C. H. Lindsey, L. G. L. T. Meertens, C. H. A. Koster, M. Sintzoff, J. E. L. Peck and R. G. Fisker, 'Revised Report on the Algorithmic Language ALGOL 68', *Acta Informatica*, **5**, 1-236 (1975).
6. 'Ada Programming Language', ANSI/MIL-STD-1815A, American National Standards Institute, New York, NY, February 1983.

7. A. H. J. Sale, 'A Note on Scope, One-Pass Compilers, and Pascal', *Pascal News*, **15**, 62-63 (1979).

8. S. L. Graham, W. N. Joy and O. Roubine, 'Hashed Symbol Tables for Languages with Explicit Scope Control', *SIGPLAN Notices*, **14**, 50-57 (August 1979).

9. N. Wirth, *Programming in Modula-2*, Springer Verlag, Heidelberg, 1985. Third Edition.